# Problem A. Orthogonal Circles

| | |
|---|---|
| Input file: | `stdin` |
| Output file: | `stdout` |
| Time limit: | 1 second |
| Memory limit: | 256 megabytes |

Two circles are *orthogonal*, if they intersect and for any point of intersection their tangent lines at that point are perpendicular.

Consider a set of $n$ circles on the plane. You are to find a circle orthogonal to all of them.

## Input

The input file contains an integer $n$ ($1 \le n \le 10^5$), followed by $n$ triples of integers: $x_i$, $y_i$, $r_i$, denoting the center coordinates and the radii of the circles ($-10^6 \le x_i, y_i \le 10^6$, $1 \le r_i \le 10^6$).

The circles may coincide.

## Output

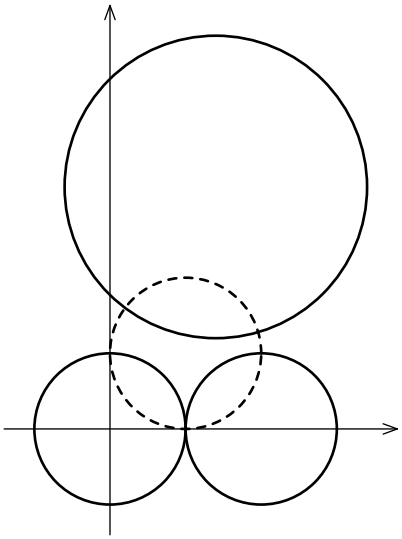
If there exists exactly one circle orthogonal to all the given ones, output its center coordinates and radius as real numbers separated with single spaces. The numbers will be considered correct if they are within $10^{-6}$ relative or absolute error of the exact answers.

In case there's no such circle, output $-1$. In case there are many, output $-2$.

## Example

| stdin | stdout |
|---|---|
| 3<br>0 0 5<br>7 16 10<br>10 0 5 | 5.0 5.0 5.0 |

The example is illustrated by the following picture:



# Problem B. Necessary Coins

| | |
|---|---|
| Input file: | stdin |
| Output file: | stdout |
| Time limit: | 1 second |
| Memory limit: | 256 megabytes |

Vasya has been on vacation on Mars. He's a big fan of foreign coins, and thus has collected exactly one martian coin of each denomination, for a total of $n$ coins: $a_1$ martian dollars, $a_2$ martian dollars, etc, $a_n$ martian dollars. Unfortunately, he couldn't stand ordering the Pan Galactic Gargle Blaster at the Starport, and has to pay for it — it costs $x$ martian dollars. Vasya is wondering which of his coins are absolutely necessary to do so (i.e., he is forced to abandon them). They don't offer change at the Starport Mars.

## Input

The input file contains two integer numbers $n$ and $x$ ($1 \le n \le 200$, $1 \le x \le 10^4$), followed by $n$ distinct integer numbers $a_i$ ($1 \le a_i \le x$).

## Output

On the first line of output, print the amount of denominations of coins that appear in any subset that sums to $x$ martian dollars. On the second line of output, print the denominations themselves, in any order, separated with single spaces. It is guaranteed that there exists at least one way to pay $x$ martian dollars with the given coins.

## Example

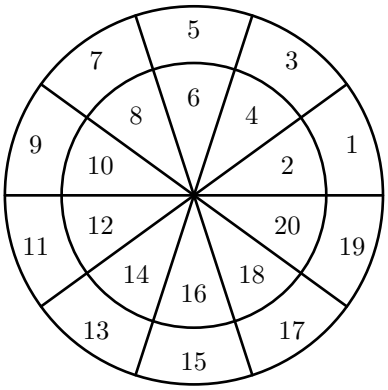
| stdin | stdout |
|---|---|
| 5 18<br>1 2 3 5 10 | 2<br>5 10 |

# Problem C. Optimal Dartboard

| | |
|---|---|
| Input file: | `stdin` |
| Output file: | `stdout` |
| Time limit: | 0.5 seconds |
| Memory limit: | 256 megabytes |

A dartboard is a disc divided into $n$ segments, with each segment labeled with a distinct positive integer between 1 and $n$. That integer indicates the score for hitting the segment. To make the game more interesting, the arrangement of numbers is chosen in such a way that the risk is maximized. The risk is estimated based on the differences in scores of adjacent segments.

We're studying the following 'double-layered' structure of segments in this problem:




I.e., $n$ is always even, and we split the disc into two layers of $\frac{n}{2}$ parts along the circumference. We enumerate the segments in the following manner: the first segment is some outer segment, the second segment is the corresponding inner segment, the third segment is some adjacent outer segment, etc. An example of this enumeration is shown on the picture above.

The total risk is defined as the sum of squared differences between the scores of adjacent segments. If the value assigned to segment $i$ is $a_i$, then the risk is:

$$R = \sum_{i=1}^{n} (a_i - a_{i+2})^2 + \sum_{i=1}^{n/2} (a_{2i-1} - a_{2i})^2$$

(we assume $a_{n+1} = a_1$ and $a_{n+2} = a_2$ in this formula).

You are to place the numbers from 1 through $n$ into the segments in such a way that the total risk $R$ is maximized.

## Input

The input file contains an even integer number $n$ ($6 \le n \le 100$).

## Output

Output $n$ integer numbers: $a_1, a_2, \ldots, a_n$, separated with single spaces. In case there are several possible solutions, output any.

## Example

| stdin | stdout |
|---|---|
| 10 | 2 9 7 4 6 5 3 8 10 1 |

# Problem D. Heavy Disc

| | |
|---|---|
| Input file: | `stdin` |
| Output file: | `stdout` |
| Time limit: | 0.5 seconds |
| Memory limit: | 256 megabytes |

Consider a heavy disc on the plane, centered at $(x_0, y_0)$ with radius $r$, with the origin strictly outside it. The density of the disc is given by formula

$$\rho(x, y) = \ln(x^2 + y^2)$$

What is the mass of the disc?

## Input

The input file contains three integer numbers $x_0$, $y_0$, $r$ ($-100 \le x_0, y_0 \le 100$, $1 \le r \le 100$, $x_0^2 + y_0^2 > r^2$).

## Output

Output one real number — the mass of the disc. Your answer will be considered correct if it is within $10^{-12}$ relative error of the exact answer.

## Example

| stdin | stdout |
|---|---|
| 3 4 2 | 40.449586576894895 |

# Problem E. Deducing Grammar

| | |
|---|---|
| Input file: | stdin |
| Output file: | stdout |
| Time limit: | 0.5 seconds |
| Memory limit: | 256 megabytes |

A typical top-down parser is a very simple application. We will consider parsers written in Pascal that obey the following structure:

```
Program Parser;
Procedure Skip; Forward;
Function Peek:Char; Forward;
Procedure Error; Forward;

<parsing routines forward-declarations>
<parsing routines>

Var
  St:String;
  Pos:Integer;
Procedure Error;
Begin
  WriteLn('NO');
  Halt;
End;
Procedure Skip;
Begin
  Inc(Pos);
  If Pos>Length(St) Then Error;
End;
Function Peek:Char;
Begin
  Peek:=St[Pos];
End;
Begin
  ReadLn(St);
  St:=St+'#';
  Pos:=1;
  Parse;
  If Pos=Length(St) Then WriteLn('YES') Else WriteLn('NO');
End.
```

The part labeled `<parsing routines forward-declarations>` contains a definition for each of the parsing functions, followed by `Forward;`. The part labeled `<parsing routines>` contains a definition followed by a body for each of the parsing routines.

A definition of a parsing routine is:

```
Procedure <name>;
```

The name is a case-insensitive sequence of English letters. Names of different routines should be different. A routine name cannot coincide with a keyword, nor can it coincide with the names of parser's internal functions (`Skip`, `Peek` or `Error`). A body of a parsing routine is:

```
Begin
  <segment>
  <segment>
  ...
  <segment>
End;
```

There must be at least one segment. Each segment is either an unconditional segment, or a conditional segment.

An unconditional segment is a routine call: `<name>;`, where `<name>` is a name of some parsing routine.

A conditional segment has the following structure:

```
If Peek=<character> Then Begin
  Skip;
  <segment>
  <segment>
  ...
  <segment>
End Else If Peek=<character> Then Begin
  Skip;
  <segment>
  <segment>
  ...
  <segment>
End Else If Peek=<character> Then Begin
...
End Else If Peek=<character> Then Begin
End;
```

Where each `<segment>` is again either unconditional or conditional. The simplest form of a conditional segment has only one `If` and no `Else`. Note that we always skip a character after guessing it right, and that's the only case where we invoke `Skip;`. Each `<character>` is a non-apostrophe character with ASCII code between 33 and 126 wrapped in apostrophes, like `'a'`. It is not necessary to have at least one `<segment>` after `Skip;` in the `If` body. The last `End;` can also be written as `End Else Error;`, meaning that all other peek outcomes are unacceptable.

The case of the letters in the identifiers and keywords can be arbitrary, and whitespace may be inserted and/or omitted everywhere except it must be present between two words and it must not be present inside a word or string literal (something wrapped in apostrophes). One of the parsing routines must be named `Parse`.

Such a top-down parser is usually based on some formal grammar. Your task is to find which grammar the given parser is based on. To do so, you should apply the following rules (note that the resulting grammar may not necessarily define the same language as the language accepted by the parser — you shouldn't care about it, just blindly apply the rules):

- The set of nonterminal symbols of the grammar is the same as the set of parsing routine names in the parser.
- The set of terminal symbols of the grammar is the set of characters with ASCII codes between 33 and 126, except apostrophe.
- For each (even seemingly impossible, like duplicate `Peek=...` conditions) way of evaluating all `If`'s in the body of some routine that do not result in a call to `Error;`, there should be one production rule concatenating the corresponding nonterminals and terminals (see example for further clarification).
- The starting symbol should be `Parse`.

Your program should input the top-down parser code in Pascal, and output the grammar in the Backus-Naur form. See example for more information on how to output the grammar. Two adjacent string literals should be concatenated, i.e., you should write `'AB'` instead of `'A''B'`. Your output should contain exactly $n$ lines, where $n$ is the number of parsing routines in the input.

The nonterminals must be written in lowercase. The production rules inside a line must be sorted lexicographically, and the lines must be sorted lexicographically, too.

## Input

The input file contains a top-down parser in Pascal. The size of the input file does not exceed 10000 bytes, and each word is at most 20 characters long.

## Output

Output the Backus-Naur form of the underlying grammar. The only whitespace in output should be line breaks after each line (including the last line), as the output will be checked for exact equality with the reference answer. The output is guaranteed not to exceed 10000 bytes.

## Example

File `stdin`:

```
Program Parser;
Procedure Skip; Forward;
Function Peek:Char; Forward;
Procedure Error; Forward;

Procedure Parse; Forward;
Procedure Addend; Forward;
Procedure Term; Forward;
Procedure Number; Forward;

Procedure Term;
Begin
  If Peek='0' Then Begin
    Skip;
    Number;
  End Else If Peek='1' Then Begin
    Skip;
    Number;
  End Else If Peek='(' Then Begin
```

```
    Skip;
    Parse;
    If Peek=')' Then Begin
      Skip;
    End Else Error;
  End Else If Peek='P' Then Begin
    Skip;
    If Peek='I' Then Begin
      Skip;
    End Else If Peek='E' Then Begin
      Skip;
    End Else Error;
  End Else Error;
End;

Procedure Number;
Begin
  If Peek='0' Then Begin
    Skip;
    Number;
  End Else If Peek='1' Then Begin
    Skip;
    Number;
  End;
End;

Procedure Addend;
Begin
  Term;
  If Peek='*' Then Begin
    Skip;
    Addend;
  End Else If Peek='/' Then Begin
    Skip;
    Addend;
  End;
End;

Procedure Parse;
Begin
  Addend;
  If Peek='+' Then Begin
    Skip;
    Parse;
  End Else If Peek='-' Then Begin
    Skip;
    Parse;
  End;
End;
```

```
Var
  St:String;
  Pos:Integer;
Procedure Error;
Begin
  WriteLn('NO');
  Halt;
End;
Procedure Skip;
Begin
  Inc(Pos);
  If Pos>Length(St) Then Error;
End;
Function Peek:Char;
Begin
  Peek:=St[Pos];
End;
Begin
  ReadLn(St);
  St:=St+'#';
  Pos:=1;
  Parse;
  If Pos=Length(St) Then WriteLn('YES') Else WriteLn('NO');
End.
```

File `stdout`:

```
<addend>::=<term>|<term>'*'<addend>|<term>'/'<addend>
<number>::=|'0'<number>|'1'<number>
<parse>::=<addend>|<addend>'+'<parse>|<addend>'-'<parse>
<term>::='('<parse>')'|'0'<number>|'1'<number>|'PE'|'PI'
```

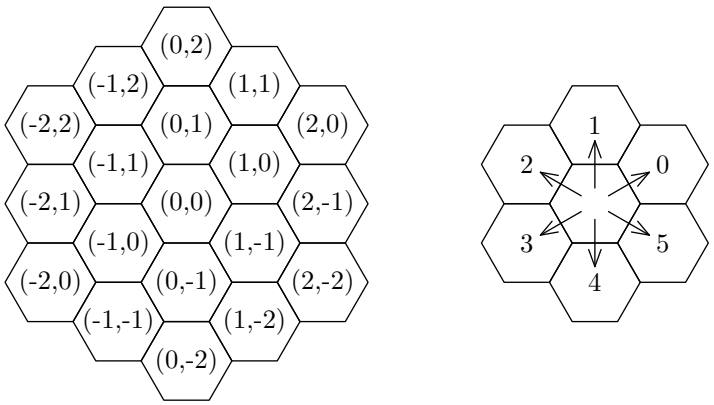
Note that this parser and this grammar correspond to arithmetic expressions involving binary numbers and two constants `'PE'` and `'PI'`.

# Problem F. Hexagonal Walkaround

| | |
|---|---|
| Input file: | stdin |
| Output file: | stdout |
| Time limit: | 0.5 seconds |
| Memory limit: | 256 megabytes |

A hexagonal leftist tank is a very complicated piece of machinery. It lives on a hexagonal grid, moving from a cell to some of its adjacent cells each turn.

More specifically, each moment of time the tank is located in some cell $(x, y)$ of the grid, heading in one of six directions (see picture for the definition of coordinate system and directions). In one turn, the tank can either move forward in the direction it is heading, or turn left one direction (i.e., increase the direction it is heading by one modulo six), and then move forward in the new direction the tank is heading. Moreover, it is forbidden to move more than $b$ times in the same direction consecutively.




You need to put the tank into cell $(x, y)$, heading in direction $d_2$, given it is located in cell $(0, 0)$ initially, heading in direction $d_1$. What is the minimal number of turns required to do that?

## Input

The input file contains five integer numbers $x$, $y$, $d_2$, $d_1$, $b$ ($-10^{12} \le x, y \le 10^{12}$, $0 \le d_1, d_2 \le 5$, $2 \le b \le 10$).

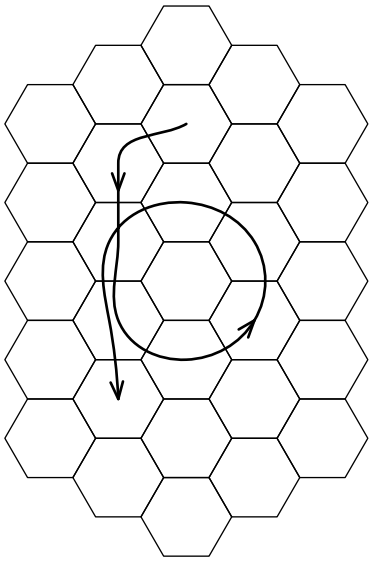
## Output

Output one integer number — the minimal number of turns required to get the tank to the required position.

## Example

| stdin | stdout |
|---|---|
| 3 3 1 5 3 | 6 |
| -1 -3 4 2 2 | 10 |

The second example is solved as follows (see next page):



# Problem G. Number Permutations

| | |
|---|---|
| Input file: | stdin |
| Output file: | stdout |
| Time limit: | 1 second |
| Memory limit: | 256 megabytes |

Two distinct positive integer numbers with decimal notations of the same length are called *similar*, if their decimal notations can be obtained from each other by permuting the digits.

How many numbers from the segment $[l, r]$ have exactly one similar number in that segment?

## Input

The input file contains two integer numbers $l$ and $r$ ($1 \le l \le r \le 10^{15}$).

## Output

Output one integer number — the sought amount.

## Example

| stdin | stdout |
|---|---|
| 10 99 | 72 |

# Problem H. k-th Product

| | |
|---|---|
| Input file: | stdin |
| Output file: | stdout |
| Time limit: | 1 second |
| Memory limit: | 256 megabytes |

Consider $n$ integer numbers $a_1, a_2, \ldots, a_n$. Consider the products of their $N = C_n^m = \frac{n!}{(n-m)! \cdot m!}$ $m$-tuples: $b_1, b_2, \ldots, b_N$. You are to find $k$-th largest product, i.e., $k$-th item in the list $(b_i)$ sorted in non-increasing order (items of the list are numbered starting from 1).

Note that the numbers can be negative.

## Input

The input file contains three integer numbers $n$, $m$, $k$ ($1 \le n, k \le 10000$, $1 \le m \le 13$, $k \le C_n^m$), followed by $n$ integer numbers $a_i$ ($-10^6 \le a_i \le 10^6$).

## Output

Output one integer number — the $k$-th largest product.

## Example

| stdin | stdout |
|---|---|
| 4 3 3<br>2 3 3 5 | 30 |

There are four possible products in the example: $3 \cdot 3 \cdot 5 = 45$, $2 \cdot 3 \cdot 5 = 30$ (involving the first 3), $2 \cdot 3 \cdot 5 = 30$ (involving the second 3), $2 \cdot 3 \cdot 3 = 18$.

# Problem I. Fast Typing

| | |
|---|---|
| Input file: | stdin |
| Output file: | stdout |
| Time limit: | 0.5 seconds |
| Memory limit: | 256 megabytes |

Vasya has little experience in typing, thus he has to look at the keyboard to locate the necessary keys, and still makes typos while doing so. For simplicity sake, we'll assume that the only type of typo he makes is replacing a character by another one. To correct those, he employs the following strategy: from time to time, he looks at the screen, and if there is any typo in the text, he removes all the characters from the end of the text to the first typo he made inclusive (i.e., he leaves only the correct part of the text intact) by pressing 'backspace' key several times, and continues typing from that position again.

Pressing any key (including 'backspace') takes 1 unit of time, and looking at the screen takes $t$ units of time. Given the probabilities of making an error for each character in the text, compute the minimal possible expected time to type the entire text correctly (including verifying that by looking at the screen in the end and noticing no typos).

The text is $n$ characters long, and the probability of mistyping $i$-th character is equal to $a_i$.

## Input

The input file contains two integer numbers $n$ and $t$ ($1 \le n \le 3000$, $1 \le t \le 10^6$), followed by $n$ real numbers $a_i$ ($10^{-5} \le a_i \le \frac{1}{2}$).

## Output

Output one real number — the minimal possible expected time. Your answer will be considered correct if it is within $10^{-6}$ relative error of the exact answer.

## Example

| stdin | stdout |
|---|---|
| 3 1<br>0.00001 0.5 0.00001 | 8.000080000800008 |